COBB

INSIDE VISUAL dBASE™

Tips & techniques for Visual dBASE and dBASE for Windows

December 1996 • Vol. 3 No. 12
US $8.50

Lock 'em up!

Back in the old days—the time of the release of dBASE IV 1.0, to be exact—dBASE gained an important new feature: the ability to create network-aware, multi-user applications *without* your having to buy a special network version or LAN pack. Every copy of dBASE IV, and every copy of each subsequent version of dBASE, has included network support. In this article, we'll discuss some of the issues in the debate about the best way to manage data in a multi-user environment.

Automatic and manual record and file locking

The solution to most multi-user dilemmas is called *locking*. You apply a lock to one or more records or to an entire table, thereby preventing other users from modifying (or, in some cases, even viewing) the data until you release the lock. Your dBASE applications can employ automatic locking, where dBASE handles all of the details of locking records and files, or you can choose to take responsibility for explicitly coding all locking operations yourself. To explain how locking operations work, we'll review the essentials of multi-user programming and present examples demonstrating when you should let dBASE perform locking tasks automatically and when you should handle locking tasks yourself.

SET CROWD ON

Writing multi-user applications isn't particularly difficult, but it does require diligence. In a multi-user environment, your application can no longer assume exclusive access to tables, and must deal properly with conflicts of interest concerning data access. As soon as you add a second user to your system, you must put in place a whole new range of protections to ensure that you maintain the integrity of the data.

The classic multi-user conflict occurs when two users try to edit the same record at the same time. This situation is unacceptable, of course—either dBASE or your application must ensure that only the first user to access the record can edit it.

Another problem arises with any operation that processes multiple records. Consider, for example, a simple AVERAGE command. In a multi-user environment, you must be sure that no user is updating records in the table as you execute the AVERAGE command. Otherwise, the resulting average may be out of date as soon as you calculate it!

Picking your lock

dBASE supports three kinds of locks: record locks, file locks, and exclusive locks. You need to understand how each works in order to write well-behaved, multi-user applications. Let's take a look at the different locks and the commands that affect them.

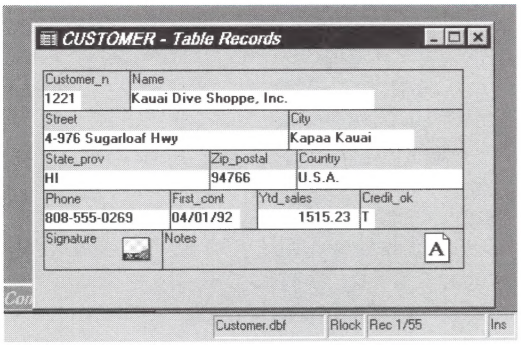
Record locks

A *record lock* prevents other users from modifying the data in the locked record

IN THIS ISSUE

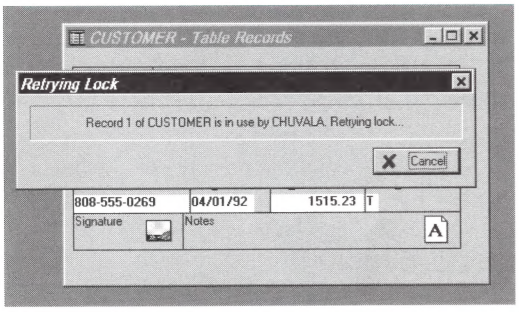
until the lock is released. You commonly use record locks to protect editing functions from multi-user conflicts. dBASE automatically attempts a record lock any time a user edits a record in a BROWSE or EDIT screen, or on a form that uses data-linked controls. In addition, a user can request a record lock (or even toggle an existing lock off) at any time by pressing [Ctrl]L while using one of those screens. When you're working with BROWSE and EDIT, the Rlock indicator in the status bar lets you know that a record lock is in effect for the current record, as shown in Figure A.

Figure A



The Rlock indicator in the status bar tells the user that a record lock is preventing others from editing the record.

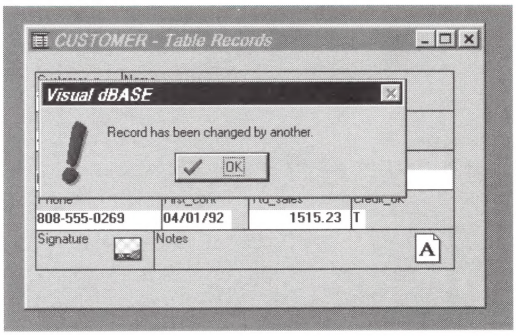
Figure B



This message will stay onscreen until the first user releases the record lock, or until other users press the Cancel button.

When a user tries to edit a record that another user has locked, dBASE presents a dialog box like the one shown in Figure B, indicating that another user has ownership of the record. dBASE retries the lock while this dialog box is onscreen. If the first user releases the lock, the dialog box will disappear. But what if the first user changes the data in the record during the lock? Won't the second user's display be out of date? Fortunately not—dBASE senses the change, and will alert you to it with a message like the one shown in Figure C.

Figure C



If the previous holder of the record lock changed the data that you want to edit, dBASE will alert you to the change.

File locks

A *file lock* prevents other users from modifying any data in the table until the lock is released. You typically use file locks to prevent changes to data during reporting and aggregate functions (AVERAGE, COUNT, SUM, and so on). The messages dBASE displays when a user tries to lock a record or file that another user has already locked resemble the record-lock messages we discussed earlier.

dBASE's SET LOCK command controls the automatic application of file locks. When SET LOCK is ON, dBASE automatically tries a file lock when it executes any of the following commands: AVERAGE, CALCULATE, COPY, COPY MEMO, COPY STRUCTURE, COPY TO ARRAY, COUNT, JOIN, LABEL FORM, REPORT FORM, SORT, SUM, or TOTAL. When SET LOCK is OFF, dBASE doesn't attempt any file locks.

Exclusive locks

An *exclusive lock* prevents all other users from accessing a table. Only the workstation that obtains the exclusive lock can work with the table. You must use an exclusive lock in order to PACK, REINDEX, or otherwise modify the structure of a table. No one else can even USE the table when you invoke an exclusive lock. If another user tries to open the table, dBASE displays a message like the one shown in Figure D.

SET REPROCESS

Now that you know what kinds of locks dBASE provides, you're ready to take a look at how to use them in your applications. For instance, suppose you want your application to control the length of time a user can wait for access to a record or file.

You can use the SET REPROCESS command to tell dBASE to try these record and file locks more than one time before reporting an error message. The maximum number of retries is 32,000. Keep in mind, though, that dBASE will appear to be unresponsive as it retries the lock—so use this command with care. You don't want users to think your application has crashed when, in fact, it's retrying a record or file lock.

When you SET REPROCESS to 0, dBASE will retry the lock indefinitely and will display the dialog box in **Figure B**. On the other hand, when you set this command to -1, dBASE will try the lock only once, and will *not* display the dialog box. This distinction becomes important when you write applications that manage locks manually, as we'll discuss shortly.

SET REFRESH

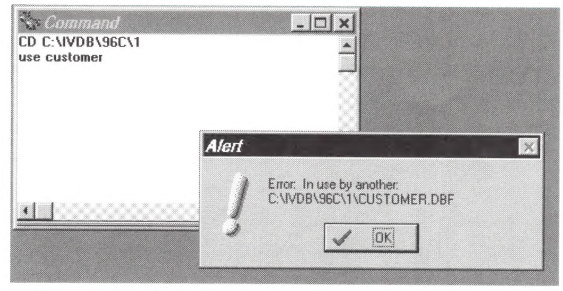
If you use the BROWSE, EDIT, and CHANGE commands interactively on a network, you can use the SET REFRESH command to control how frequently dBASE refreshes the current data display. You simply set the interval to a number of seconds between 1 and 3600, inclusive. For example, the command

```
SET REFRESH TO 10
```

tells dBASE to refresh the data onscreen every 10 seconds. If you set the interval to 3600, dBASE will refresh the screen once every hour. You should be careful about using the SET REFRESH command, though: The refresh process can interrupt your editing if you happen to be typing new data at the same time dBASE is repainting the screen!

You can perform "manual" refreshing in a couple of ways. Form objects have a Refresh( ) method, but its behavior is somewhat unreliable. The SET REFRESH command, however, works fine both in the interactive dBASE environment and within form-based code. We'll show you a form that uses this command later.

Figure D



When another user has an exclusive lock on a table, you can't even USE it!

CONVERTing tables

The CONVERT command adds a field called _DBASELOCK to a table. This special field gives dBASE a place to store locking information that you can exploit in your applications. The _DBASELOCK field stores the date and time of the current (or last) lock, as well as the user ID of the person who obtained the lock. A change count also resides in this field, and dBASE uses that count when it executes the CHANGE( ) function.

I'm a great believer in CONVERTing tables, but I have one warning to offer: If you access your dBASE tables with other dBASE-compatible products, you may sometimes find that those products can't deal with the _DBASELOCK field. In such cases, the foreign applications may display a message telling you the table is *not a dBASE-compatible file*, which is, of course, nonsense! But it's easy to understand why some dBASE-compatible products get confused. The name of the field isn't a valid dBASE field name, since you can't begin a normal field name with the underscore character (_).

When you CONVERT a table, dBASE creates a copy of the table—with a CVT extension— in the same directory as the original table. If you change your mind about the CONVERT process, you can rename the CVT file DBF and then open it normally. Note that if you issue a command in the form

```
USE FILENAME.CVT
```

Visual dBASE will display an *illegal filename* error.

Locking with forms

So far, we've talked about the automatic locking that dBASE offers. But what about real applications in which you, the developer, want more control over behavior and error messages? Fortunately, dBASE offers a number of useful functions that make it relatively easy for you to take control. Let's explore these functions and how you can use them.

SET EXCLUSIVE ON or OFF

When you set EXCLUSIVE to ON, dBASE opens all tables with an exclusive lock. This behavior is rarely what you want in a multi-user environment. You can always employ a command in the form *USE tablename EXCLUSIVE* for reindexing, packing, and other activities that require an exclusive lock. In general, you'll want your applications to default to SET EXCLUSIVE OFF.

Rlock(record[s],"alias")

You'll probably use the Rlock( )function most often in a multi-user environment. This function asks dBASE for a record lock on the specified record(s) in the specified alias—or on the current record in the current work area, if you don't pass the function any arguments. Rlock( ) returns a logical value indicating whether dBASE could grant the lock.

Rlock( ) will honor the SET REPROCESS setting. A high setting will sometimes make your application appear to hang while the program retries the lock. With SET REPROCESS TO 0, Rlock( ) will retry the lock indefinitely and present the dialog box shown in **Figure B**. With the setting SET REPROCESS TO -1, Rlock( ) will try the lock once, no dialog box will appear, and the function will immediately return the result (.T. or .F.). I typically use the latter setting in my applications. If I can't get the lock immediately, the application will ask whether to retry or cancel, or whatever other options might be appropriate for the application.

Rlock( ) sometimes does more work than you intend it to. When a parent-child relation is in effect (set up via a query or a SET RELATION command), locking a record in the parent table also locks any child records specified by SET RELATION and SET SKIP. This makes good sense, of course, and saves you a lot of work. (For more information about parent-child relationships, see "Establishing Good Relations" in the November 1996 issue of *Inside Visual dBASE*.)

Flock("alias")

Flock( ) asks dBASE to obtain a file lock on the specified alias, or on the table in the current work area when no alias is specified. The dynamics of Flock( ) are similar to those of Rlock( )—again, you need to pay attention to SET REPROCESS's status when you use Flock( ) in your applications.

UNLOCK

The UNLOCK command releases any file and/or record lock(s) applied in the current work area. You can use UNLOCK ALL to release all locks in all work areas, or you can specify a particular work area with a command in the form

```
UNLOCK IN alias_name
```

to release locks only in that area. DBASE automatically releases record locks on the current record when the record pointer moves off that record.

What, no Elock( )?

You may be surprised to learn that there isn't a built-in Elock( ) function that tells you whether you can obtain an exclusive lock. To work around this situation, you can code your own function or write some in-line code that accomplishes the task. Here's an example:

```
* Elock() - tries to get exclusive use of a file,
* returns .F.if unsuccessful, or .T. if
* successful
*
function Elock
   parameter cFilename
   on error nError = 1
   nError = 0
   use (cFilename) exclusive
return (nError == 0)
```

Note that this function sets the ON ERROR trap for its own use, so you'll have to reset ON ERROR after using the Elock( ) function.

A demonstration form

Listing A presents LOCKEM.WFM, which illustrates record-locking within a form. The program actually creates two forms (see Figure E) that use a sample table. LOCKEM will create the table for you the first time you run the form.

Examine the code and the behavior of the form. I've SET REPROCESS to 0 in the program so that it will be very obvious when dBASE tries to obtain record locks. As an exercise, you might try modifying the form to perform all lock management manually.

Note the use of the REFRESH command rather than the Form.Refresh( ) method for the Refresh pushbutton. As mentioned earlier, the behavior of Form.Refresh( ) is a bit erratic in Visual dBASE version 5.5a; stick with the REFRESH command, instead.

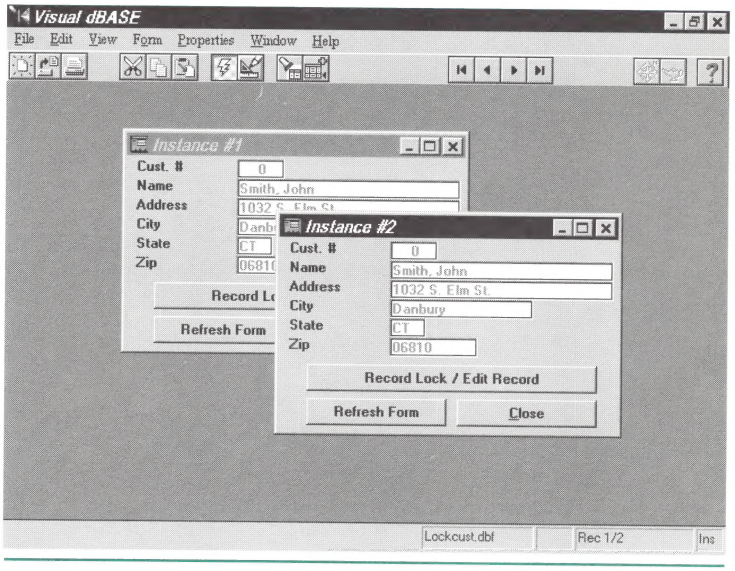
LOCKEM.WFM demonstrates the principles described in this article—even on a standalone machine—as long as the Borland Database Engine (BDE) is set up correctly. You must ensure that the LOCALSHARE option is set to TRUE. Run the BDE Configuration Utility application that ships with Visual dBASE, and see Figure F for the details. If you don't have an icon for the BDE Config program, you must run it manually. You can usually find it by running C:\IDAPI\BDECONFIG.EXE.

Even better, run the demonstration form over a network, with two or more stations accessing it from the same network directory. You'll see how dBASE picks up the user's name from his login name (it can also use a name that's been entered through the PROTECT security system), and more.

It's a lock!

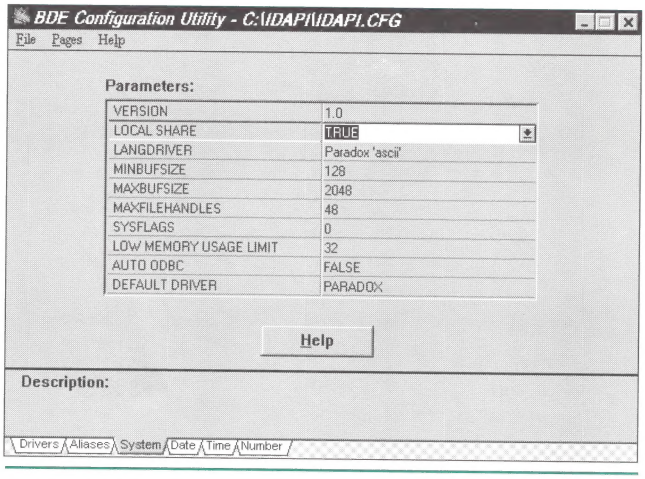
You must understand the fundamentals of record and file locking in order to write a good multi-user application. I'm interested in hearing your multi-user and network challenges and problems. Network environments are constantly increasing in complexity. Let me know how these changes are affecting your development environment and the applications you need to write. Send me E-mail at **kgc@jinx.sckans.edu** or via CompuServe at **71333,2654**. ❖

Figure E



LOCKEM.WFM displays these two forms.

Figure F



Set the LOCALSHARE option to TRUE using the BDE Configuration Utility.

Listing A: *LOCKEM.WFM*

```
* LOCKEM.WFM
*

shell(.f.,.t.)

* Create a lockcust table if one does not exist,
* then CONVERT it for multi-user functions.
if .not. file("lockcust.dbf")
   msgbox("Demo table LOCKCUST.DBF will be created","")
   create table "lockcust.dbf" ( ;
        id      numeric(5,0),;
        name    char(30), ;
        address char(35), ;
        city    char(35), ;
```

Continued on page 6

Continued from page 5

```
        state   char(2), ;
        zip     char(10) )
   use lockcust exclusive
   convert to 16
   append blank
   replace id      with 0, ;
           name    with "Smith, John", ;
           address with "1032 S. Elm St.",;
           city    with "Danbury",;
           state   with "CT", ;
           zip     with "06810"
   append blank
   replace id      with 1, ;
           name    with "Jones, Alice", ;
           address with "#5 Easy St.",;
           city    with "Winfield",;
           state   with "KS", ;
           zip     with "67156"

   use
endif

* Create a couple of sessions and open a couple of forms

create session
set reprocess to 0
set refresh to 0
f1 = new lockemform()
f1.text = "Instance #1"
f1.open()

create session
set reprocess to 0
set refresh to 0
f2 = new lockemform()
f2.text = "Instance #2"
f2.top = f1.top + 2
f2.left = f1.left + 10
f2.open()

return


** END HEADER * do not remove this line*
* Generated on 07/15/96
*
parameter bModal
local f
f = new lockemFORM()
if (bModal)
   f.mdi = .F. && ensure not MDI
   f.ReadModal()
else
   f.Open()
endif
CLASS lockemFORM OF FORM
   this.Text = "LockEm.WFM"
   this.OnOpen = {; form.controls(.f.)}
   this.OnClose = {; unlock ; close databases}
   this.Height = 10
   this.Width = 50
   this.Left = 8
   this.Top = 2
   this.CanNavigate = CLASS::CHECKSTATUS
   this.OnNavigate = CLASS::CHECKSTATUS
   this.View = "lockcust.dbf"

   DEFINE TEXT TEXT1 OF THIS;
       PROPERTY;
         Height 1,;
         Width 13.166,;
         Text "Cust. #",;
         Left 1.5,;
         Top 0.1172

   DEFINE ENTRYFIELD EF_ID OF THIS;
       PROPERTY;
         DataLink "lockcust->ID",;
         Height 1,;
         Width 7,;
         Left 16.333,;
         Top 0.1172

   DEFINE TEXT TEXT2 OF THIS;
       PROPERTY;
         Height 1,;
         Width 13.166,;
         Text "Name",;
         Left 1.5,;
         Top 1.1172

   DEFINE ENTRYFIELD EF_NAME OF THIS;
       PROPERTY;
         DataLink "lockcust->NAME",;
         Height 1,;
         Width 33,;
         Left 16.333,;
         Top 1.1172

   DEFINE TEXT TEXT3 OF THIS;
       PROPERTY;
         Height 1,;
         Width 13.166,;
         Text "Address",;
         Left 1.5,;
         Top 2.1172

   DEFINE ENTRYFIELD EF_ADDRESS OF THIS;
       PROPERTY;
         DataLink "lockcust->ADDRESS",;
         Height 1,;
         Width 33,;
         Left 16.333,;
         Top 2.1172

   DEFINE TEXT TEXT4 OF THIS;
       PROPERTY;
         Height 1,;
         Width 13.166,;
         Text "City",;
         Left 1.5,;
         Top 3.1172

   DEFINE ENTRYFIELD EF_CITY OF THIS;
       PROPERTY;
         DataLink "lockcust->CITY",;
         Height 1,;
         Width 21.167,;
```

```
      Left 16.333,;
      Top 3.1172

DEFINE TEXT TEXT5 OF THIS;
    PROPERTY;
      Height 1,;
      Width 13.166,;
      Text "State",;
      Left 1.5,;
      Top 4.1172

DEFINE ENTRYFIELD EF_STATE OF THIS;
    PROPERTY;
      DataLink "lockcust->STATE",;
      Height 1,;
      Width 5.333,;
      Left 16.333,;
      Top 4.1172

DEFINE TEXT TEXT6 OF THIS;
    PROPERTY;
      Height 1,;
      Width 13.166,;
      Text "Zip",;
      Left 1.5,;
      Top 5.1172

DEFINE ENTRYFIELD EF_ZIP OF THIS;
    PROPERTY;
      DataLink "lockcust->ZIP",;
      Height 1,;
      Width 13,;
      Left 16.333,;
      Top 5.1172

DEFINE PUSHBUTTON PB_RLOCK OF THIS;
    PROPERTY;
      Height 1.4111,;
      Width 42.835,;
      Group .T.,;
      Text "Record Lock / Edit Record",;
      Left 4.165,;
      Top 6.5293,;
      OnClick CLASS::PB_RLOCK_ONCLICK

DEFINE PUSHBUTTON PB_REFRESH OF THIS;
    PROPERTY;
      Height 1.4121,;
      Width 20.668,;
      Group .T.,;
      Text "Refresh Form",;
      Left 4.165,;
      Top 8.293,;
      OnClick {; Refresh}

DEFINE PUSHBUTTON PUSHBUTTON1 OF THIS;
    PROPERTY;
      Height 1.4121,;
      Width 20.668,;
      Group .T.,;
      Text "&Close",;
      Left 26.332,;
      Top 8.293,;
      OnClick {; Unlock; form.close()}
```

```
Procedure PB_RLOCK_OnClick
   if left(this.text,1)="R"
      if rlock()
         this.text = "Unlock it!"
         form.controls(.t.)
      else
         msgbox("Lock NOT Successful!","Rlock() Failed")
      endif
   else
      form.saverecord()
      form.do_unlock()
   endif

procedure do_unlock
   form.controls(.f.)
   unlock
   form.pb_rlock.text = "Record Lock / Edit Record"

*
* Enable/disable editing controls depending on lock status
*
procedure controls
   parameter lEnable
   form.refresh()
   form.ef_id.enabled      = lEnable
   form.ef_name.enabled    = lEnable
   form.ef_address.enabled = lEnable
   form.ef_city.enabled    = lEnable
   form.ef_state.enabled   = lEnable
   form.ef_zip.enabled     = lEnable
   if lEnable
      form.ef_id.setfocus()
   endif

procedure CheckStatus
   if form.isrecordchanged()
      #define YesNo 4
      #define Yes   6
      if msgbox("Save changes?","Data has changed!",YesNo) == Yes
         form.saverecord()
      else
         form.abandonrecord()
      endif
   endif
   form.do_unlock()

ENDCLASS
```

FUNCTION TIP

Getting the message across

Most Windows applications use dialog boxes and message boxes to notify the user of actions in progress and to prompt the user to make a choice. If you thought it would take a lot of programming to display a window with a custom message and an OK button, you're in for a nice surprise. In this article, we'll show you how you can use a single line of code to add a message box to your Visual dBASE or dBASE for Windows application.

The MSGBOX( ) function

In its simplest form, the MSGBOX( ) function interrupts your program and displays a message and an OK pushbutton in a window centered onscreen. The message window remains onscreen until the user presses [Enter] or clicks the OK button. Then, the function returns control to your program.

You use the MSGBOX( ) function much as you would a supercharged version of the WAIT command in dBASE for DOS. If you want to get the user's attention and you require acknowledgment of an onscreen message, then you'll use the MSGBOX( ) function.

You call the MSGBOX( ) function with a single command in the form

```
result = MSGBOX("message")
```

where *message* is the text you want to display. For instance, suppose you want to display a warning when the operator enters an invalid value in a field. You simply add the following command to create the message box and display it onscreen:

```
result = MSGBOX("Sorry, try again!")
```

This command displays a message box in the middle of the screen, centering the text of your message and displaying an OK button below the message, as shown in Figure A.

Figure A

You can display a message box with a single line of code.

Figure B

Our message box now features a custom title.

Adding a title to the message box

As you can see in Figure A, the default message box's title area is blank. If you want to add your own title, you can pass it as the second parameter, also a string value, when you call MSBOX( ). You use a command in the form

```
result = MSGBOX("message", "title")
```

where *message* and *title* are the custom message and title you want to display. To illustrate, let's add a title to our *Sorry, try again!* message box by entering the command

```
result = MSGBOX("Sorry, try again!",
    "Friendly Computing Services")
```

This command displays the message box shown in Figure B. Again, the function waits for the operator to press [Enter] or click OK before returning control to the calling program.

When you call the MSGBOX( ) function as part of a replacement statement, as we've just done, the function returns the value 1 when the user clicks the OK button. You can demonstrate this behavior by entering the command *? result* after calling the MSGBOX( ) function. When TALK is ON, dBASE will display the results onscreen, as shown in Figure C.

Creating more complex message boxes

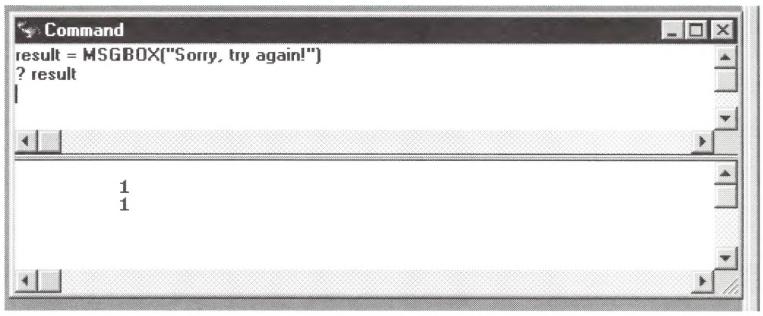
When you call the MSGBOX( ) function and pass only two parameters—a message and a title—the function serves for acknowledg-

ment purposes only. That is, it displays the message, gets a response from the operator, and then returns to your program.

By adding a third parameter—display type—when you call MSGBOX( ), you can include special icons and different sets of pushbuttons in your message box. In "Enhancing Message Boxes with Icons and Pushbutton Sets," on page 13, we'll show you how you can display predefined icons and pushbutton sets in a custom MSGBOX( ) call.

When you add the third parameter, you can use the MSGBOX( ) as more than simply a tool for forcing the user to acknowledge a message. When the message box contains different pushbutton sets, the function returns different values, depending on which button the user presses. You can use those values to control what your program does after the user responds to the message box. ❖

Figure C



The MSGBOX( ) function returns a value of 1 when the user clicks the OK button.

FORM TIP

Using legacy programs with forms

John W. Pogue, a dBASE programmer from Columbia, Tennessee, contributed the material on which we based this article. You can send John E-mail at ***johnpogue@aol.com****.*

We've received a number of letters from former and current dBASE for DOS programmers requesting help in leveraging their DOS applications in the Windows environment. In this article, we'll present an application that demonstrates the basics of programming a form in Visual dBASE. We'll show you how to

- use the DataLink property to create a dropdown list that uses an array for its data source
- create a form field that stores a user-supplied date in a variable
- use the DataSource property to create a calculated field on your form that calls an external PRG file to return a value

If you want, you can use the code for our demonstration form as the basis for your own custom forms.

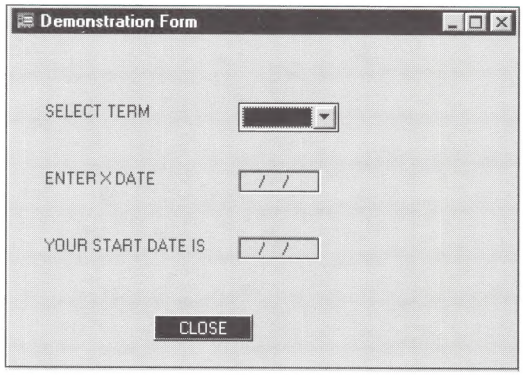
Forming expectations

Suppose you've written a dBASE for DOS program (a PRG file) that performs a specific calculation—such as determining the start date of an insurance policy—and you want to use that program to calculate the same results in a Visual dBASE form (a WFM file). Fortunately, all you must know is how to tell the form to call your PRG.

To demonstrate our technique, we'll create a form with three fields and a Close button, like the one shown in Figure A. One field lets you choose an option from a dropdown list, one lets you enter a date value, and one displays the result of a calculation, based on the entries in the first two fields.

When you click the field beside the label *SELECT TERM*, the dropdown list of options appears, as

Figure A

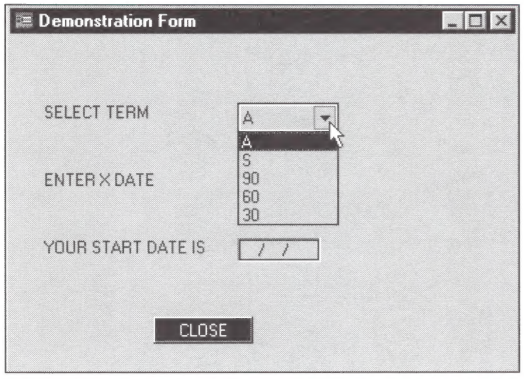


We'll create a sample form like this one.

shown in Figure B. You can choose an item by clicking on it, by highlighting it and pressing [Enter], by pressing ↑ or ↓, or by typing the first character (letter or numeral) of the option.

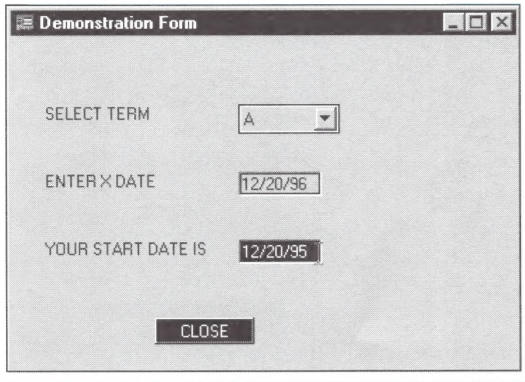
After choosing an option from the dropdown list, you press [Tab] to move to the field labeled *ENTER XDATE*. After entering a date, press [Enter] to move to the *YOUR START DATE IS* field. Then, the form automatically calculates the start date, as shown in Figure C.

Figure B



A dropdown list makes it easy for you to choose a valid option.

Figure C



When you enter the final field, the form automatically calculates the start date.

Looking behind the scenes

Now that you've seen what our form looks like, let's take a look at the code behind it. First, the code that displays the form itself appears in Listing A. You can create this file by entering *MODIFY COMMAND ST_DATE.WFM* in the Command window and then typing the code. You can also create the form using the Form Designer. Then, use any text editor to open and customize the resulting WFM file.

After you create the standalone PRG files TE_ARRAY and ST_DATE, run the form by entering

```
DO ST_DATE.WFM
```

at the Command window. Now, let's take a look at the first line and the third-from-last line in the listing.

You'll notice we've emphasized two filenames—TE_ARRAY and ST_DATE—by printing them in green. That way, if you use the code in Listing A as a model for your own form, you'll know exactly where to insert calls to your own programs.

You'll find the code for TE_ARRAY.PRG in Listing B; the code for ST_DATE.PRG appears in Listing C. TE_ARRAY.PRG simply defines an array that stores the possible options for the first field in our form. ST_DATE.PRG, on the other hand, performs some custom calculations used by Mr. Pogue to determine start dates for his insurance clients, based on a given expiration date.

Creating the dropdown list

Providing dropdown lists to let the user choose an option is a popular form technique for two reasons. First, end users don't have to memorize options—they simply have to be able to choose the appropriate option from a list. Second, using the dropdown options prevents users from entering an invalid entry.

You create the dropdown list with the section of code

```
DEFINE COMBOBOX COMBOBOX1 OF THIS;
```

We've used the term *dropdown list* to refer to the form object Visual dBASE calls a *combobox*. The command

```
DataSource "ARRAY TE_ARRAY",;
```

tells the form to look in the array TE_ARRAY for the values to use in the dropdown list. When you create your own dropdown lists, you simply substitute the appropriate array name in the place of *TE_ARRAY*. As you may know, dBASE provides several other options for the

source of data for your dropdown list, such as a database file. However, using arrays provides a faster way to display the data, since dBASE accesses the array directly from memory.

Placing the calculated field on the form

In the same way you define a calculated field in a report or for a BROWSE screen, so can you create a field on a form that performs a custom calculation. In a form, you use the DataLink property to tell the form what value to display. For instance, in ST_DATE.WFM, the command

```
DEFINE ENTRYFIELD ENTRYFIELD2 OF THIS;
```

defines the entry field in which the user enters a date. The command

```
DataLink "mX_DATE",;
```

tells dBASE to store the value the user enters in a variable named mX_DATE.

Then, the command

```
DEFINE ENTRYFIELD ENTRYFIELD3 OF THIS;
```

defines a field named ENTRYFIELD3, and the command

```
DataLink "mST_DATE",;
```

tells the form to display the value of the variable mST_DATE. But where does mST_DATE come from? The answer lies in the following section of code from the end of ST_DATE.WFM:

```
Procedure ENTRYFIELD2_OnLostFocus
      mTERM = FORM.COMBOBOX1.VALUE
      DO ST_DATE.PRG WITH mX_DATE,mTERM
      FORM.ENTRYFIELD3.VALUE = mST_DATE
```

In this case, ENTRYFIELD2 is the second field on the form—the field in which the user enters a date. When the OnLostFocus event fires for this field—which it does when you press [Tab] to leave the field—the form executes the subsequent commands. Specifically, the command

```
mTERM = FORM.COMBOBOX1.VALUE
```

assigns to the variable mTERM the user's selection from the combobox's dropdown list. Then, the command

```
DO ST_DATE.PRG WITH mX_DATE,mTERM
```

calls the ST_DATE.PRG external procedure, passing the parameters that program requires. Finally, the command

```
FORM.ENTRYFIELD3.VALUE = mST_DATE
```

updates the third field on our form with the result that ST_DATE.PRG returns.

Summing up

When you want to leverage an existing program in a Visual dBASE form, you can leave your code in standalone PRG files, or you can copy your PRG into the WFM file and call it as a procedure. In this article, we've shown you the basics of how to use the DataSource property to populate an array, and how to use the DataLink property to display on your form the value an external program returns. ❖

Listing A: *ST_DATE.WFM*

```
DO TE_ARRAY.PRG
PUBLIC mX_DATE,mTERM,mST_DATE
mX_DATE = {}
mTERM   = SPACE(2)
mST_DATE = {}
** END HEADER * do not remove this line*
* Generated on 08/11/96
*
parameter bModal
local f
f = new ST_DATEFORM()
if (bModal)
   f.mdi = .F. && ensure not MDI
   f.ReadModal()
else
   f.Open()
endif
CLASS ST_DATEFORM OF FORM
   this.ColorNormal = "GB"
   this.Left = 21
   this.Text = "Form"
   this.Top = 0
   this.Height = 14
   this.Width = 60

   DEFINE TEXT TEXT1 OF THIS;
       PROPERTY;
```

Continued on page 12

Continued from page 11

```
        ColorNormal "B/GB",;
        Left 4,;
        Text "SELECT TERM",;
        Top 2.8223,;
        Height 0.7637,;
        Width 15,;
        FontBold .F.

  DEFINE COMBOBOX COMBOBOX1 OF THIS;
      PROPERTY;
        ColorNormal "W+/B*",;
        Left 27,;
        Top 2.8223,;
        Height 1.1768,;
        DataSource "ARRAY TE_ARRAY",;
        Width 12,;
        FontBold .F.,;
        Style 2

  DEFINE TEXT TEXT2 OF THIS;
      PROPERTY;
        ColorNormal "B/GB",;
        Left 4,;
        Text "ENTER X DATE",;
        Top 5.6465,;
        Height 0.7637,;
        Width 15,;
        FontBold .F.

  DEFINE ENTRYFIELD ENTRYFIELD2 OF THIS;
      PROPERTY;
        ColorNormal "N/W",;
        ColorHighLight "W+/B*",;
        OnLostFocus CLASS::ENTRYFIELD2_ONLOSTFOCUS,;
        Left 27,;
        Top 5.6465,;
        Height 1,;
        DataLink "mX_DATE",;
        Width 9.8311,;
        FontBold .F.

  DEFINE TEXT TEXT3 OF THIS;
      PROPERTY;
        ColorNormal "B/GB",;
        Left 4,;
        Text "YOUR START DATE IS",;
        Top 8.4697,;
        Height 0.7646,;
        Width 20,;
        FontBold .F.

  DEFINE ENTRYFIELD ENTRYFIELD3 OF THIS;
      PROPERTY;
        ColorNormal "N/W",;
        ColorHighLight "W+/B*",;
        Left 27,;
        Top 8.4688,;
        Height 1,;
        DataLink "mST_DATE",;
        Width 9.8311,;
        FontBold .F.

  DEFINE PUSHBUTTON PUSHBUTTON1 OF THIS;
      PROPERTY;
        ColorNormal "B/GB",;
        Group .T.,;
        Left 17,;
        Text "CLOSE",;
        Top 11.7637,;
        Height 1.1172,;
        Width 11.832,;
        Tabstop .F.,;
        OnClick {;RELEASE mX_DATE,mTERM,
➥       mST_DATE,TE_ARRAY; FORM.CLOSE()},;
        FontBold .F.

  Procedure ENTRYFIELD2_OnLostFocus
     mTERM = FORM.COMBOBOX1.VALUE
     DO ST_DATE.PRG WITH mX_DATE,mTERM
     FORM.ENTRYFIELD3.VALUE = mST_DATE
ENDCLASS
```

Listing B: *TE_ARRAY.PRG*

```
* TE_ARRAY.PRG
* Initialize array for lookup table for TERM field
PUBLIC TE_ARRAY
DECLARE TE_ARRAY[1,5]
STORE "A"   TO TE_ARRAY[1,1]
STORE "S"   TO TE_ARRAY[1,2]
STORE "90"  TO TE_ARRAY[1,3]
STORE "60"  TO TE_ARRAY[1,4]
STORE "30"  TO TE_ARRAY[1,5]
```

Listing C: *ST_DATE.PRG*

```
* ST_DATE.PRG
* Enter your own program here.
...
* Return the result your program computes.
* e.g., RETURN mST_DATE
*- EOP ST_DATE() **********
```

Enhancing message boxes with icons and pushbutton sets

The article "Getting the Message Across," on page 8, describes how to use the MSGBOX( ) function to display a message box with a custom message and optional title. In this article, we'll show you how to use the MSGBOX( ) function's display type parameter to enhance your message box with icons and pushbutton sets.

Colorful icons provide a visual cue to end users indicating the significance of your message boxes. The pushbutton sets provide built-in return values that correspond to the buttons, making your message box a versatile programming tool.

A quick review

You can pass the MSGBOX( ) function up to three parameters—message, title, and display type. In its simplest form, you use the function to display a message box by passing only the first parameter with a call like

```
result = MSGBOX("message")
```

where *message* is the text you want to display. To add a title to the message box, enter a command in the form

```
result = MSGBOX("message", "title")
```

where you replace *message* and *title* with the message and title for your custom message box.

The display type parameter

The real power of the MSGBOX function is built into the optional display type parameter. This parameter is a combination of codes that tells dBASE what icon to display in the message box and what set of pushbuttons to offer the operator. You call the function using all three possible parameters with a command in the form

```
result = MSGBOX("message", "title",
    display_type_code)
```

where *message* and *title* contain the appropriate strings, and you replace the parameter *display_type_code* with the sum of the numbers corresponding to the icon you want to use and to the pushbutton set you want to include in your message box.

Choosing an icon

In Windows applications, you use icons to give end users a visual cue about the importance of the message you're displaying. For instance, you might use a bright red exclamation point to indicate the most urgent type of message. On the other hand, you might use the lowercase *i* to identify a message you display for information purposes only.

As Table A shows, you can customize your message box by choosing from among four icons: a red exclamation point, a green question mark, a yellow exclamation point, and a blue *i*. You simply jot down for later use the number that corresponds to the icon you select.

Table A: *Message box icons and display code values*

Icon	Add to Display Code Value
Red Exclamation Point	16
Green Question Mark	32
Yellow Exclamation Point	48
Blue Lowercase i	64

Table B: *Message box button set options*

Button Set	Add to Display Code Value
OK	0
OK, Cancel	1
Abort, Retry, Ignore	2
Yes, No, Cancel	3
Yes, No	4
Retry, Cancel	5

Selecting a button set

You use pushbutton sets to allow an operator to decide what should happen next at a certain point in the program. You choose a set of pushbuttons for your custom message box by determining the set of choices that you want to give the operator. For example, if your message box asks the

question, *Are you sure you want to quit?* you can select the set of pushbuttons that allows the operator to select either Yes or No.

You'll find the six pushbutton set options in Table B, on page 13. You simply decide what set of choices you want to give the operator, and then add the corresponding value to the *display_type_code* parameter when you call the MSGBOX( ) function.

Calculating the value for the display code

Once you select the icon for the set of pushbuttons, you must determine the value to send to the MSGBOX( ) function for the display type. To calculate this value, you simply add the two numbers that represent your choices—their sum is the value you use as the third parameter when you call MSGBOX( ).

For example, suppose you want to create a message box that verifies the operator's intent to delete a record. You want to use the question mark as the icon, and a button set that contains the OK and Cancel options. Since the question mark icon's value is 32, and the OK and Cancel button set's value is 1—for a total of 33—you'd call MSGBOX( ) with a command in the form

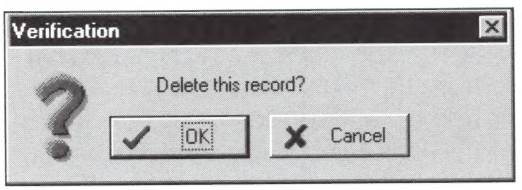
```
result = MSGBOX("Delete this record?",
    "Verification", 33)
```

When you issue this command, dBASE displays a message box like the one shown in Figure A.

Determine which button the user pushes

After you display a message box that offers the operator a choice of actions to take, you need to code a routine that will perform those actions. The MSGBOX( ) function returns a code to the calling program that identifies which button the operator selects.

Figure A



You can display this message box with one line of code.

Since there are a number of different button sets that contain different choices, dBASE assigns a standard value to each of the pushbuttons in the display. For example, three of the button sets contain a button labeled *Cancel*. Regardless of which button set you select, the Cancel button will always return the same value—2. Table C shows the values the standard buttons return.

Table C: *Values MSGBOX( ) returns*

When you press Pushbutton	MSGBOX( ) returns
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

Using MSGBOX( ) in a routine

Now that you know how the MSGBOX( ) function works, let's use it in a program. Suppose you want to prompt the operator to confirm a deletion by choosing OK or Cancel. Once the MSGBOX( ) function returns a value that corresponds to the operator's choice, you can use a DO CASE command to perform the appropriate action. The code for such a routine might take the form

```
* Code that locates the record
* for possible deletion.
result = ;
MSGBOX("Delete this record?", "Verification",33)
DO CASE
    CASE result=1       && Operator pressed OK
        DELETE
    CASE result=2    && Operator pressed Cancel
        RETURN
ENDCASE
```

If the operator chooses OK, the function returns a value of 1. The CASE statement *result=1* evaluates to True, and dBASE executes the DELETE command. When the operator selects Cancel, the function returns a value of 2, and the program executes the RETURN command. ❖

Customizing the message box

In the accompanying article, we show you how to use the MSGBOX( ) function to display custom messages and action options for your end users. By default, when MSGBOX( ) creates a window, it determines the window's size based upon the button set, icon, and message you want to display. Sometimes, however, you'll want to override the default size and format.

When you pass a particularly long message to the MSGBOX( ) function, the function wraps the text into a window in the center of the screen. To illustrate, let's create a message box that advises the operator that the data entered is invalid and that only six codes are acceptable. Enter the command

```
result = MSGBOX("Invalid Code " +  ;
"Allowable Codes  include"+"A - Apple " + ;
"B - Banana"+"O - Orange"+"P - Pineapple "+ ;
"R - Raisins"+"T - Tangerine", + ;
"Error Processing", 16)
```

When you execute this command, dBASE wraps the message text onto a second line.

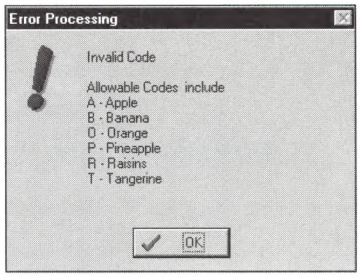
Fortunately, you can force the function to break the string of message text at specific points by inserting the expression CHR(13)—a carriage return character—into your string of message text. Just place this expression at each point where you want to start a new line. As you'd expect, the carriage return character starts a new line, regardless of the amount of text remaining in the message.

To illustrate, let's modify our previous function call so that it displays each option on a separate line. To do so, you'd embed the CHR(13) expression in the form

```
result = MSGBOX("Invalid Code "+ CHR(13) +
    ;
CHR(13) + "Allowable Codes  include" +;
CHR(13) + "A - Apple " + CHR(13) +;
"B - Banana " + CHR(13) + "O - Orange" +;
CHR(13) + "P - Pineapple " + CHR(13) +;
"R - Raisins"+ CHR(13) +;
"T - Tangerine", + "Error Processing", 16)
```

When dBASE executes this command, it displays a message box like the one shown in Figure A. ❖

Figure A



You can override the default word wrap by embedding carriage return characters in the message string.

INSIDE VISUAL dBASE™

Inside Visual dBASE (ISSN 1084-1970) is published monthly by The Cobb Group.

Staff:
Contributing Editor-in-Chief ..... Keith G. Chuvala
Associate Editors-in-Chief ..... Jeff E. Davis
Tiffany M. Taylor
Publications Coordinator ..... Maureen Spencer
Editors ..... Laura Merrill
Joan McKim
Elisabeth Pehlke
Production Artist ..... Alison Schwarz
Product Group Manager ..... Mike Stephens
Circulation Manager ..... Mike Schroeder
VP/Publisher ..... Mark Crane
President ..... John A. Jenkins

Address:
Please send tips, special requests, and other correspondence to

The Editor, *Inside Visual dBASE*
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

E-mail address: visual_dbase_win@merlin.cobb.zd.com

For subscriptions, fulfillment questions, and requests for group subscriptions, address your letters to

Customer Relations
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

E-mail address: customer_relations@merlin.cobb.zd.com

Phone:
Toll free US ..... (800) 223-8720
Toll free UK ..... (0800) 961897
Local ..... (502) 493-3300
Customer Relations Fax ..... (502) 491-8050
Editorial Department Fax ..... (502) 491-3433

Back Issues:
To order back issues, call Customer Relations at (800) 223-8720. Back issues cost $8.50 each, $8.95 outside the US. You can pay with MasterCard, VISA, Discover, or American Express, or we can bill you.

Postmaster:
Periodical postage paid in Louisville, KY.

Postmaster: Send address changes to:

Inside Visual dBASE
P.O. Box 35160
Louisville, KY 40232

Prices:
Domestic ..... $79/yr ($8.50 each)
Outside US ..... $89/yr ($8.95 each)

Borland Technical Support
Technical Support
Information Line: (800) 523-7070
TechFAX System: (800) 822-4269

QUICK TIP

Making and changing directories from within your program

Beginning with dBASE for Windows 5.0, Borland introduced two powerful commands in the dBASE language: MKDIR and CD. As in DOS, these two commands allow you to make directories and change directories, respectively.

The MKDIR command

You can use the new MKDIR command in your programs to create new directories on the fly without leaving dBASE. The command works the same way that it does in DOS. That is, it takes the form

```
MKDIR path
```

where *path* is the pathname you want to create. For example, suppose you want to create a subdirectory called SAMPLES that exists under the current working directory. To do so, issue a command in the form *MKDIR SAMPLES*. This command creates a new subdirectory under the current directory. As with DOS, if you put a backslash (\) in front of the directory name, the MKDIR command will create the new directory under the root directory of the current drive. For instance, the command *MKDIR \SAMPLES* creates a directory named *SAMPLES* off the root directory.

Using MKDIR for installations

The MKDIR command can help when you want to set up directories without leaving your dBASE application. For example, suppose you have a special program that installs a system for the first time. You want to make sure you have certain directories present before proceeding with the installation. You can quickly create the new directories and subdirectories using the MKDIR command.

If the directory you're creating already exists, dBASE will give you an error message. In addition, all the other standard DOS constraints about creating directories apply when you use this command in your dBASE application.

The CD command

In dBASE IV, you change the working directory by using the SET DIRECTORY, SET DEFAULT, and SET PATH commands. In dBASE for Windows and Visual dBASE, you can change the current directory by using the standard DOS command CD in your programs.

The CD command follows all of the standard DOS conventions for moving from one directory to another, including the use of the double period. For instance, when you issue the new CD command in the form

```
CD..
```

the program will make current the directory at the next higher level in the directory tree.

Conclusion

By including the MKDIR and CD.. commands in your programs, you can maintain better control over the location of your files. You can use these commands to simplify the task of managing directory-level file functions in your applications. ❖